Práctica 5: Algoritmos meméticos
Selección de características
Algoritmos meméticos con búsqueda local generalizada, aleatoria y elitista

Alejandro García Montoro
76628233F, agarciamontoro@correo.ugr.es

Grupo de los viernes a las 17.30

Curso 2015 - 2016

Índice

1. Descripción del problema

La selección de características es una técnica muy usada en problemas de aprendizaje automático.

El aprendizaje automático, visto de una forma muy general, tiene como objetivo clasificar un conjunto de objetos —modelado por una serie de atributos— en clases.

Esta clasificación se aprende desde los datos, pero la selección de los atributos que definen la modelización del objeto puede no ser la más apropiada: en ocasiones hay atributos superfluos o demasiado ruidosos que sería conveniente eliminar. Además, cuantos menos atributos definan un objeto, más rápido y preciso será el aprendizaje. Es aquí entonces donde aparece la pregunta que guía todo este trabajo: ¿cómo identificar los atributos que mejor aprendizaje promueven?

La respuesta a esta pregunta pasa por la selección de características, cuyo objetivo es reducir la definición de un objeto a una serie de características que faciliten el aprendizaje.

La idea es entonces la siguiente: dado un conjunto de m objetos definidos por un conjunto C de n características, y considerado un modelo de aprendizaje f que intenta aprender la clasificación de estos objetos, encontrar el subconjunto $C' \subset C$ que maximiza el modelo f.

Así, vemos claramente que el tamaño de caso de nuestro problema es n —el número de características— y que el objetivo está bien definido: eliminar aquellas características que o bien empeoren la bondad de f o bien sean innecesarias.

Con todos estos elementos definidos, podemos pasar a analizar las metaheurísticas consideradas.

2. Metaheurísticas

2.1. Introducción

Los algoritmos considerados para resolver el problema son todos algoritmos meméticos, que hibridan un algoritmo genético generacional con la búsqueda local de las tres formas siguientes:

- Generalizado (AM1010): Cada 10 generaciones, se ejecuta una iteración del algoritmo de búsqueda local primero el mejor sobre todos los cromosomas que forman la población.
- Aleatorio (AM1001): Cada 10 generaciones, se ejecuta una iteración del algoritmo de búsqueda local primero el mejor sobre un 10 % de los cromosomas elegido de forma aleatoria.
- Elitista (AM1001): Cada 10 generaciones, se ejecuta una iteración del algoritmo de búsqueda local primero el mejor sobre el mejor 10 % de los cromosomas.

Además, compararemos estas metaheurísticas con el algoritmo voraz *Sequential forward selection*.

Estas tres metaheurísticas reúnen las condiciones necesarias para resolver el problema: el espacio de soluciones de nuestro problema puede ser analizado mediante las estructuras de generación de vecinos y los criterios de aceptación que utilizan estos algoritmos. Veamos con un poco más de detalle los aspectos comunes a las metaheurísticas implementadas:

Datos de entrada

Todos los algoritmos considerados reciben un conjunto de entrenamiento cuyos objetos tienen la siguiente estructura:

$$(s_1, s_2, \ldots, s_n, c)$$

donde $(s_1, s_2, \ldots, s_n)$ es el conjunto de valores de los atributos que definen el objeto y c la clase a la que pertenece.

Esquema de representación

El espacio de soluciones S de nuestro problema es el conjunto de todos los vectores s de longitud n —el número de características— binarios; es decir:

$$S = \{s = (s_1, s_2, \ldots, s_n)/s_i \in \{0, 1\}\ \forall i = 1, 2, \ldots, n\}$$

La posición i-ésima de un vector $s \in S$ indicará la inclusión o no de la característica i-ésima en el conjunto final C'.

Función objetivo

La finalidad de las metaheurísticas será maximizar la función objetivo siguiente:

$$\begin{aligned} f\colon S &\to [0,100] \\ s &\mapsto f(s) = \text{Acierto del 3-NN sobre s} \end{aligned}$$

$f(s)$ es, por tanto, la tasa de acierto del clasificador 3-NN producido a partir de la solución s.

El clasificador 3-NN es una particularización del clasificador k-NN, que mide la distancia de la instancia considerada a todos los demás objetos en el conjunto de datos de entrenamiento y le asigna la clasificación mayoritaria de entre los k vecinos más cercanos; esto es:

Pseudocódigo 1 Clasificador k-NN

```
1: function k-NN(instance, trainingData)
2:     distances ← euclideanDistance(instance, trainingData)
3:     neighbours ← getClosestNeighbours(distances)
4:     classification ← mostVotedClassification(neighbours)
5:     return classification
```

Así, dada una solución $s \in S$, la función objetivo es como sigue:

Pseudocódigo 2 Función objetivo

```
1: function f(s, train, target)
2:     samples ← removeZeroColumns(s, train)
3:     sum ← 0
4:     for instance ∈ samples do
5:         class ← k-NN(instance, samples)
6:         sum ← sum + { 1  if class = actualClass(instance, target)
                       { 0  if class ≠ actualClass(instance, target)
7:     return sum / (number of samples in train)
```

donde *removeZeroColumns(s, train)* elimina la columna i-ésima de *train* si y sólo si $s_i = 0$ y *actualClass(instance, target)* devuelve la clase real —no la aprendida— del objeto *instance*.

Entorno de soluciones

Dada una solución $s \in S$, el entorno de soluciones vecinas a s es el conjunto

$$E(s) = \{s' \in S / |s' - s| = (0, \ldots, 0, \underbrace{1}_{i}, 0, \ldots, 0), i \in \{1, 2, \ldots, n\}\}$$

es decir, $E(s)$ son las soluciones que difieren de s en una única posición. Es evidente entonces que el conjunto $E(S)$ tiene siempre exactamente cardinal igual a n.

El operador de generación de vecino de la solución s es entonces como sigue:

Pseudocódigo 3 Operador de generación de vecino

```
1: function FLIP(solution, feature)
2:     s' ← solution
3:     s'[feature] ← (s'[feature] + 1) mod 2
4:     return s'
```

Criterios de parada

En todos los algoritmos pararemos tras ejecutar 15000 evaluaciones con el clasificador 3-NN sobre las soluciones generadas.

Búsqueda local primero el mejor

El algoritmo de búsqueda local usado para la hibridación es el algoritmo primero el mejor, visto en la primera práctica. La única diferencia con aquel es que en este caso se ejecuta una única iteración del mismo, haya habido mejora o no.

El pseudocódigo de todo el procedimiento es el siguiente:

Pseudocódigo 4 Búsqueda local primero el mejor

```
1: function BESTFIRST(chromosome)
2:     s ← chromosome
3:     bestScore ← score(s)
4:     for f ← genRandomFeature(s) do          ▷ Without replacement
5:         s'← flip(s,f)
6:         score ← score(s')
7:         if score > bestScore then
8:             bestScore ← score
9:             s ← s'
10:            break
11:    return s, bestScore
```

Mecanismo de selección

En el algoritmo genético se considera un mecanismo de selección basado en el torneo binario; es decir, se eligen dos individuos de la población al azar y se selecciona el mejor. En el Pseudocódigo 5 se puede ver este procedimiento:

Pseudocódigo 5 Torneo binario

```
1: function BINARYTOURNAMENT(population)
2:     contestants ← randomly pick 2 chromosomes from population
3:     winner ← best(contestants_1, contestantes_2)
4:     return winner
```

El mecanismo de selección generacional puede verse en el Pseudocódigo 6.

Pseudocódigo 6 Mecanismo de selección

```
1: function SELECTION(population)
2:     for i ∈ {1, 2, ..., n} do                ▷ n = size of the population
3:         s_i ← binaryTournament(population)
4:     selected ← (s_1, s_2, ..., s_n)
5:     return selected
```

Operador de cruce

El operador de cruce clásico en dos puntos consiste en lo siguiente: dividir los dos padres en tres partes iguales para ambos individuos pero de tamaño aleatorio y asignar a cada hijo toma la parte central de un padre y las partes exteriores del otro. Podemos ver este operador en el Pseudocódigo 7.

Pseudocódigo 7 Operador de cruce clásico

```
1: function CLASSICXOVER(f, m)                ▷ Father and mother
2:     i, j ← pick 2 random integers in {2, ..., n − 1} ▷ n = number of genes
3:     c^1 ← (f_1, f_2, ..., f_i, m_{i+1}, m_{i+2}, ..., m_j, f_{j+1}, f_{j+2}, ..., f_n)
4:     c^2 ← (m_1, m_2, ..., m_i, f_{i+1}, f_{i+2}, ..., f_j, m_{j+1}, m_{j+2}, ..., m_n)
5:     children ← [c^1, c^2]
6:     return children
```

Operador de mutación

Por último, estudiemos el operador de mutación considerado, dependiente de un proceso también aleatorio. La idea inicial era la siguiente: para cada gen de cada cromosoma se genera un número aleatorio; si este es menor que una constante α, se muta el gen; si no, se deja tal y como está.

Como la constante considerada es ínfima —$\alpha = 0{,}001$—, el coste computacional de generar un número aleatorio para cada gen de cada individuo de la población es muy alto: generaremos demasiados valores aleatorios para las pocas mutaciones que vamos a realizar. Por tanto, se ha seguido un procedimiento basado en el número esperado de mutaciones; es decir: se calcula el

número $M = \alpha n N$, donde α es la probabilidad de mutación, n el número de genes en un cromosoma y N el número de cromosomas en la población y se eligen M genes de entre todos los cromosomas a los que se le aplica la mutación. El operador de mutación atómico —esto es, el procedimiento que se le aplica a cada gen si se decide mutarlo— es el operador *flip*. En el Pseudocódigo 8 se puede ver todo este proceso con más detalle, donde se indica que los números aleatorios generados en las M iteraciones deben ser siempre distintos para no mutar un mismo cromosoma con un mismo gen dos veces.

Pseudocódigo 8 Operador de mutación

```
1:  function MUTATE(population)
2:      α ← 0,001
3:      n ← size of a chromosome
4:      N ← number of chromosomes in population
5:      M ← ⌈αnN⌉
6:      for _ ∈ {1, 2, ..., M} do                    ▷ Repeat it M times
7:          ▷ Do not repeat the pair {chromosome, gene} between iterations
8:          chromosome ← random({1, ..., N})
9:          gene ← random({1, ..., n})
10:         flip(chrom, gene)
11:     return population
```

Esquema de evolución

La idea de la evolución es la siguiente: para cada pareja de individuos seleccionados en P_t se generará un número aleatorio; si este es menor que una constante α prefijada, a esta pareja se le aplicará el operador de cruce, guardando el resultado como parte de la población de descendientes; si el número aleatorio generado es mayor que α, la pareja de individuos seleccionados pasará a formar parte de la población de descendientes sin más.

Veamos el procedimiento general en el Pseudocódigo 9

Pseudocódigo 9 Esquema de evolución

```
1:  function RECOMBINATION(selected, α)   ▷ Selected chromosomes in P_t
2:      D ← ∅                                          ▷ Descendants of P_t
3:      for f,m ∈ pairs(selected) do ▷ Take a different pair in each iteration
4:          random ← random([0.0,1.0])
5:          D ← D ∪ { crossover(f, m)   if random < α
                    { {f, m}            if random ≥ α
6:      return D
```

Esquema de reemplazamiento generacional

El algoritmo genético sigue un esquema generacional, así que la población P_{t+1} está formada por todos los descendientes de P_t —esto es, el conjunto de individuos devuelto por el método *recombination*—, asegurando el elitismo; es decir, que la mejor solución de P_t esté en P_{t+1}. El procedimiento, que recibe como parámetros la población P_t y los descendientes D generados con el esquema de evolución recién descrito, puede verse en el Pseudocódigo 10.

Pseudocódigo 10 Reemplazamiento generacional

```
1: function genReplacement(P_t, D)        ▷ Actual population and its descendants
2:     M ← best chromosome in P_t
3:     P_{t+1} ← D
4:     if M ∉ P_{t+1} then
5:         W ← worst chromosome in P_{t+1}
6:         P_{t+1} ← (P_{t+1} \ {W}) ∪ {M}
7:     return P_{t+1}
```

2.2. Algoritmo memético con búsqueda local generalizada

Con todos los procedimientos explicados, estamos ya en disposición de ver el comportamiento de los algoritmos meméticos.

El primero de ellos es el que hace una iteración de la búsqueda local primero el mejor sobre todos y cada uno de los cromosomas de la población cada diez generaciones. En el Pseudocódigo 11 se encuentra resumido el procedimiento, cuyo criterio de parada depende del número de llamadas a la función objetivo: cuando se alcancen 15000 —contando aquí las llamadas hechas dentro de la búsqueda local— se detendrá el proceso y se devolverá la mejor solución.

2.3. Algoritmo memético con búsqueda local aleatoria

El segundo algoritmo considerado es el que hace una iteración de la búsqueda local primero el mejor sobre un 10 % aleatorio de los cromosomas de la población cada diez generaciones. En el Pseudocódigo 12 se encuentra resumido el procedimiento, cuyo criterio de parada depende del número de llamadas a la función objetivo: cuando se alcancen 15000 —contando aquí las llamadas hechas dentro de la búsqueda local— se detendrá el proceso y se devolverá la mejor solución.

Es importante describir el porqué de la decisión de tomar directamente un 10 % de los cromosomas de la población: el algoritmo a desarrollar está descrito a través de una probabilidad —$p_{LS} = 0{,}1$— que determina para cada cromosoma de la población si este se debe mejorar con la búsqueda

Pseudocódigo 11 Algoritmo memético generalizado

```
1: function AGG
2:     α ← 0,7                                          ▷ Crossover probability
3:     N ← 10                                           ▷ Size of the population
4:     P_t ← generate N random chromosomes
5:     while calls to the target function <15000 do
6:         S ← selection(P_t, N)                        ▷ Selection of N parents
7:         D ← recombination(S, α)
8:         D' ← mutation(D)
9:         P_{t+1} ← genReplacement(P_t, D')
10:        if generation % 10 == 0 then
11:            for C ∈ P_{t+1} do
12:                C ← bestFirst(C)
13:        P_t ← P_{t+1}
14:    bestSolution ← bestChromosome(P_t)
15:    bestScore ← f(bestSolution)
16:    return bestChromosome, bestScore
```

Pseudocódigo 12 Algoritmo memético aleatorio

```
1: function AGG
2:     α ← 0,7                                          ▷ Crossover probability
3:     N ← 10                                           ▷ Size of the population
4:     P_t ← generate N random chromosomes
5:     while calls to the target function <15000 do
6:         S ← selection(P_t, N)                        ▷ Selection of N parents
7:         D ← recombination(S, α)
8:         D' ← mutation(D)
9:         P_{t+1} ← genReplacement(P_t, D')
10:        if generation % 10 == 0 then
11:            P' ← random 10 % sample of P_{t+1}
12:            for C ∈ P' do
13:                C ← bestFirst(C)
14:        P_t ← P_{t+1}
15:    bestSolution ← bestChromosome(P_t)
16:    bestScore ← f(bestSolution)
17:    return bestChromosome, bestScore
```

local o no. En vez de recorrer todos los cromosomas y generar un número aleatorio para cada uno de ellos, lo que se ha hecho es calcular el número esperado de cromosomas a mejorar localmente.

Así, basta tomar una muestra aleatoria de ese tamaño —lo que llamamos P' en el pseudocódigo— y aplicar la mejora sólo a esos cromosomas.

2.4. Algoritmo memético con búsqueda local elitista

El último algoritmo considerado es el que hace una iteración de la búsqueda local primero el mejor sobre el mejor 10 % de los cromosomas de la población cada diez generaciones. En el Pseudocódigo 13 se encuentra resumido el procedimiento, cuyo criterio de parada depende del número de llamadas a la función objetivo: cuando se alcancen 15000 —contando aquí las llamadas hechas dentro de la búsqueda local— se detendrá el proceso y se devolverá la mejor solución.

Pseudocódigo 13 Algoritmo memético elitista

```
1:  function AGG
2:      α ← 0,7                                  ▷ Crossover probability
3:      N ← 10                                   ▷ Size of the population
4:      P_t ← generate N random chromosomes
5:      while calls to the target function <15000 do
6:          S ← selection(P_t, N)                ▷ Selection of N parents
7:          D ← recombination(S, α)
8:          D' ← mutation(D)
9:          P_{t+1} ← genReplacement(P_t, D')
10:         if generation % 10 == 0 then
11:             P' ← best 10 % sample of P_{t+1}
12:             for C ∈ P' do
13:                 C ← bestFirst(C)
14:         P_t ← P_{t+1}
15:     bestSolution ← bestChromosome(P_t)
16:     bestScore ← f(bestSolution)
17:     return bestChromosome, bestScore
```

2.5. Algoritmo de comparación

Para la comparación de los algoritmos implementados consideraremos el algoritmo voraz *Sequential forward selection*, que se puede ver en el Pseudocódigo 14.

La idea es la siguiente: en cada iteración escogemos la característica, de entre las aún no seleccionadas, que mejor valor de la función objetivo produce, si y sólo si este valor es mejor que el actual.

Pseudocódigo 14 Algoritmo de comparación

```
1: function sequentialForwardSelection(train, target)
2:     s ← genZeroSolution()
3:     bestScore ← 0
4:     while there was improvement with some feature do
5:         for every feature f in not selected features do
6:             s ← addFeature(s,f)
7:             currentScore ← score(s, train, target)
8:             if currentScore > bestScore then
9:                 bestScore ← currentScore
10:                bestFeature ← f
11:            s ← removeFeature(s,f)
12:        if there was a best feature f then
13:            s ← addFeature(s,f)
14:    return s, bestScore
```

3. Desarrollo de la práctica

La práctica se ha desarrollado por completo en Python, definiendo cada algoritmo en una función diferente con cabeceras iguales —mismo número y tipo de parámetros— para poder automatizar el proceso de recogida de datos.

3.1. *Framework* de aprendizaje automático

Se ha usado, además, el módulo *Scikit-learn*, del que se ha usado la siguiente funcionalidad:

- Particionamiento de los datos. *Scikit-learn* aporta una función para hacer un particionado aleatorio de los datos en una parte de aprendizaje y otra de test. Esto se ha usado para implementar la técnica 5×2 *cross-validation*.

3.2. Paralelización en GPU de la función objetivo

Aunque en las dos primeras prácticas se usó también *Scikit-learn* para medir la función objetivo, la lentitud de este proceso me llevó a buscar otras alternativas: después de intentar usar el mismo módulo con la opción de paralelización CPU y conseguir prácticamente los mismos resultados —para notar mejoría, dicen los desarrolladores, haría falta trabajar con bases de datos con varios miles de muestras—, decidí buscar una solución propia.

Como gracias a mi Trabajo fin de grado he aprendido a hacer computación general paralelizada en GPU, decidí usar la librería CUDA —y en concreto su interfaz para Python, PyCUDA— para implementar la función objetivo de una forma eficiente. La mejoría en tiempo conseguida es muy notable —es del orden de 20 a 100 veces más rápido[1]— y, tras muchas pruebas para comprobar que el cálculo de la función era correcto, sustituí el k-NN de *Scikit-learn* con el implementado en CUDA.

Todo este trabajo, necesario para el correcto funcionamiento de la práctica, se encuentra en los ficheros bajo el directorio *src/knnGPU*, que contienen la implementación en C del k-NN y la interfaz para poder usar el código desde Python.

Además, como vi que este código podía beneficiar a mis compañeros, decidí publicarlo de forma abierta en un repositorio de Github[2], bien documentado y con una guía de uso.

Gracias a esto, algunos amigos me ayudaron a mejorar el código: yo había implementado sólo la función objetivo sobre los datos de training, y Jacinto

[1] Los tiempos son muy dependientes del número de muestras de la base de datos y del número de características. Para tener una idea de la mejora, se pueden comparar los tiempos de las tablas 3-NN y SFS de esta y la anterior práctica.

[2] `https://github.com/agarciamontoro/metaheuristics`

Carrasco Castillo la modificó para poder hacer la medición también con los datos de test. Además, Luís Suárez Lloréns me ayudó a probar cambios que creíamos que iban a mejorar aún más la eficiencia —aunque tras mucho trabajo vimos que la implementación inicial era la más rápida—. Por último, Antonio Álvarez Caballero, Anabel Gómez Ríos y Gustavo Rivas Gervilla me ayudaron a testear el código, probándolo con sus algoritmos y los datos que tenían de anteriores prácticas.

3.3. Manual de usuario

Para la ejecución de la práctica es necesario tener instalado Python 3, el módulo *Scikit-learn*, *PyCUDA* y *jinja2* —estos dos últimos módulos son necesarios para la implementación del código paralelizado—, así como disponer de una tarjeta gráfica compatible con CUDA.

Todo se encuentra automatizado en el fichero `src/05_memetic.py`, así que sólo es necesario ejecutar la siguiente orden desde el directorio raíz de la práctica: `python src/05_memetic.py`

Así se ejecutarán todos los algoritmos con todas las bases de datos usando la ténica del 5×2 *cross-validation*. Las tablas generadas se guardarán en el directorio `results/05`.

La semilla utilizada se inicializa al principio de la ejecución del programa con las líneas `np.random.seed(19921201)` y `random.seed(19921201)`.

4. Análisis de resultados

En esta sección vamos a presentar los datos recogidos de la ejecución de todos los algoritmos con las tres bases de datos consideradas: *WDBC*, *Movement Libras* y *Arrhytmia*. Las bases de datos se han considerado completas en todos los casos, tal y como se nos entregaron —arreglando alguna columna defectuosa y homogeneizando el nombre de la columna de clasificación para poder automatizar el proceso—.

Para el análisis de cada algoritmo con cada base de datos se han generado cinco particiones aleatorias de los datos y se ha ejecutado el algoritmo considerando cada partición como datos de entrenamiento y test, con la técnica 5×2 *cross-validation*.

En cada una de estas ejecuciones se han medido los siguientes datos:

- Tasa de clasificación en la partición de entrenamiento —en %—.
- Tasa de clasificación en la partición de test —en %—.
- Tasa de reducción de las características —en %—.
- Tiempo de ejecución —en segundos—.

Veamos ya los datos y analicemos los resultados obtenidos:

4.1. Clasificador k-NN

	WDBC				Movement Libras				Arrythmia			
Particiones	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
Partición 1-1	97,1831	96,4912	0	$4{,}7 \cdot 10^{-3}$	68,8889	71,1111	0	$4{,}7 \cdot 10^{-3}$	64,0625	60,3093	0	0,0201
Partición 1-2	96,4912	96,1268	0	$3{,}2 \cdot 10^{-3}$	72,2222	75,5556	0	$4{,}7 \cdot 10^{-3}$	60,8247	63,5417	0	0,0184
Partición 2-1	96,4789	96,8421	0	$3{,}3 \cdot 10^{-3}$	72,2222	73,3333	0	$4{,}7 \cdot 10^{-3}$	62,5	61,8557	0	0,0195
Partición 2-2	96,8421	97,1831	0	$3{,}4 \cdot 10^{-3}$	73,3333	75	0	$4{,}7 \cdot 10^{-3}$	60,8247	65,625	0	0,0182
Partición 3-1	96,831	96,4912	0	$3{,}3 \cdot 10^{-3}$	70	77,7778	0	$4{,}7 \cdot 10^{-3}$	61,9792	64,433	0	0,0196
Partición 3-2	97,193	96,4789	0	$3{,}7 \cdot 10^{-3}$	66,6667	76,1111	0	$4{,}7 \cdot 10^{-3}$	63,9175	63,0208	0	0,0181
Partición 4-1	95,0704	96,1404	0	$3{,}5 \cdot 10^{-3}$	68,3333	63,3333	0	$4{,}7 \cdot 10^{-3}$	64,0625	62,3711	0	0,0195
Partición 4-2	95,7895	96,4789	0	$3{,}5 \cdot 10^{-3}$	71,1111	75	0	$4{,}6 \cdot 10^{-3}$	62,3711	61,9792	0	0,0184
Partición 5-1	95,7746	95,0877	0	$3{,}3 \cdot 10^{-3}$	65,5556	77,7778	0	$4{,}7 \cdot 10^{-3}$	65,1042	61,8557	0	0,0196
Partición 5-2	96,8421	96,4789	0	$3{,}3 \cdot 10^{-3}$	73,3333	73,3333	0	$4{,}6 \cdot 10^{-3}$	60,8247	67,1875	0	0,0181
Medias	96,4496	96,3799	0	$3{,}5 \cdot 10^{-3}$	70,1667	73,8333	0	$4{,}7 \cdot 10^{-3}$	62,6471	63,2179	0	0,019

Cuadro 1: Datos del clasificador k-NN

En la tabla 1 se pueden ver los datos obtenidos del clasificador k-NN. La selección de características en este algoritmo es nula, ya que es la propia función objetivo considerando la totalidad de las características. Aún así, se ha añadido aquí para conocer la tasa de clasificación en los conjuntos de entrenamiento y de test considerando como solución la trivial: esto es, todas las características.

Como vemos, aunque en la primera base de datos las tasas de clasificación son buenas, en las otras dos son muy mejorables, lo que nos da una idea de la necesidad de la reducción de características.

4.2. Algoritmo de comparación

	WDBC				Movement Libras				Arrythmia			
Particiones	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
Partición 1-1	97,8873	95,4386	83,3333	0,2377	80	73,8889	91,1111	0,8672	83,3333	71,134	97,482	2,4778
Partición 1-2	97,5439	93,3099	80	0,2718	74,4444	67,7778	88,8889	1,0971	74,7423	63,5417	97,1223	2,7568
Partición 2-1	96,831	92,6316	80	0,2733	76,6667	73,8889	91,1111	0,8691	71,3542	56,701	98,9209	1,0899
Partición 2-2	96,8421	93,662	93,3333	0,1055	78,3333	68,8889	90	0,9758	79,3814	69,2708	98,2014	1,7464
Partición 3-1	97,1831	93,6842	90	0,1429	73,8889	71,1111	90	0,9752	79,6875	67,0103	97,1223	2,7988
Partición 3-2	97,8947	95,4225	86,6667	0,1835	78,3333	70,5556	92,2222	0,7755	83,5052	68,75	96,7626	3,2249
Partición 4-1	97,8873	96,1404	80	0,2635	78,3333	77,7778	91,1111	0,873	76,5625	74,7423	98,2014	1,7183
Partición 4-2	96,8421	92,2535	90	0,1419	75	72,2222	91,1111	0,8684	73,7113	70,3125	98,9209	1,0876
Partición 5-1	97,1831	95,0877	86,6667	0,1829	76,6667	69,4444	85,5556	1,4422	79,6875	70,1031	97,8417	2,0614
Partición 5-2	98,2456	95,0704	86,6667	0,1833	75,5556	74,4444	91,1111	0,8776	73,7113	71,3542	98,2014	1,7179
Medias	97,434	94,2701	85,6667	0,1986	76,7222	72	90,2222	0,9621	77,5677	68,292	97,8777	2,068

Cuadro 2: Datos del algoritmo *Sequential forward selection*

En la tabla 2 vemos los resultados del algoritmo de comparación, el *Sequential forward selection*. Este algoritmo voraz tiene una alta tasa de reducción de características, pero la tasa de clasificación no mejora la del clasificador con la solución trivial, excepto en la última base de datos.

Esto se debe a que consideramos cada característica de una forma secuencial, y una vez seleccionamos una, es imposible descartarla. Aún así, este algoritmo podría ser interesante si lo que buscamos es una reducción drástica del número de características —como vemos, sobre el 80 %— sin perder mucha información —las tasas de clasificación son más o menos iguales a las del clasificador con la solución trivial—.

4.3. Algoritmo memético con búsqueda local generalizada

	WDBC				Movement Libras				Arrythmia			
Particiones	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
Partición 1-1	98,5915	96,4912	30	40,555	75,5556	77,2222	53,3333	41,6497	72,9167	69,5876	57,554	111,2475
Partición 1-2	97,193	95,4225	46,6667	37,1458	71,1111	76,1111	54,4444	41,3367	73,7113	64,5833	54,6763	110,7824
Partición 2-1	97,1831	94,7368	30	40,4545	76,6667	73,8889	47,7778	44,9471	70,8333	67,0103	49,6403	127,7587
Partición 2-2	98,2456	95,7746	33,3333	40,1121	78,8889	69,4444	55,5556	41,4919	73,1959	67,7083	50,7194	109,0818
Partición 3-1	97,5352	96,1404	46,6667	36,067	73,8889	77,2222	52,2222	44,2425	68,75	61,8557	50,3597	127,6069
Partición 3-2	97,193	96,1268	60	33,0628	75,5556	71,6667	52,2222	44,1694	74,7423	67,7083	50,3597	109,3431
Partición 4-1	97,5352	95,0877	40	37,3674	80,5556	77,2222	53,3333	42,5524	72,9167	63,9175	45,6835	151,9974
Partición 4-2	97,193	94,3662	50	34,4221	76,1111	68,3333	43,3333	47,699	75,2577	64,0625	50	123,4539
Partición 5-1	97,1831	96,4912	40	37,8166	75	72,7778	52,2222	43,3635	69,2708	61,8557	52,518	114,1645
Partición 5-2	98,9474	94,7183	36,6667	38,6343	72,2222	78,3333	62,2222	38,4621	73,7113	63,0208	45,3237	124,1856
Medias	97,68	95,5356	41,3333	37,5638	75,5556	74,2222	52,6667	42,9914	72,5306	65,131	50,6835	120,9622

Cuadro 3: Datos del algoritmo memético con búsqueda local generalizada.

Veamos ya el primero de los tres algoritmos meméticos estudiados.

En general, el algoritmo memético generalizado consige unas tasas de clasificación algo mejores que el SFS —excepto en la última base de datos—, pero no demasiado; quizás la gran intensificación de la búsqueda local sobre todos los cromosomas reduce al mismo tiempo la diversidad de la población, lo que provoca que los resultados no sean mucho mejores.

Habría que estudiar qué problema tiene el algoritmo, analizando la diversidad de la población y probando alternativas, pero tal y como está ahora mismo, los resultados no son muy aceptables.

4.4. Algoritmo memético con búsqueda local aleatoria

	WDBC				Movement Libras				Arrythmia			
Particiones	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
Partición 1-1	98,5915	95,7895	40	37,8719	74,4444	77,2222	40	50,5407	68,75	63,9175	48,2014	141,3621
Partición 1-2	98,2456	94,0141	53,3333	34,5613	71,1111	71,6667	52,2222	44,2274	72,1649	69,7917	50,3597	115,4163
Partición 2-1	97,5352	94,7368	63,3333	33,1349	80	73,3333	52,2222	42,6477	73,4375	62,8866	54,3165	118,0471
Partición 2-2	97,5439	94,7183	46,6667	35,9502	75,5556	68,3333	41,1111	49,211	70,1031	69,7917	47,1223	129,3192
Partición 3-1	96,831	93,3333	53,3333	33,4508	73,3333	75	52,2222	44,2715	72,3958	66,4948	48,2014	146,1793
Partición 3-2	98,2456	96,1268	53,3333	34,9881	71,6667	71,6667	56,6667	41,698	70,6186	65,1042	55,3957	109,0341
Partición 4-1	97,5352	96,8421	46,6667	35,7158	76,6667	76,1111	50	43,4785	72,3958	65,9794	49,6403	130,2166
Partición 4-2	97,8947	95,7746	53,3333	33,8378	80	71,1111	48,8889	45,6048	72,1649	64,0625	47,8417	119,2166
Partición 5-1	97,8873	95,7895	50	34,3717	75	71,1111	60	39,8537	70,8333	63,9175	50	137,1513
Partición 5-2	98,5965	93,3099	46,6667	35,827	70	80	55,5556	42,8012	72,1649	63,5417	46,0432	123,7585
Medias	97,8907	95,0435	50,6667	34,971	74,7778	73,5556	50,8889	44,4334	71,5029	65,5488	49,7122	126,9701

Cuadro 4: Datos del algoritmo memético con búsqueda local aleatoria.

En este caso se reduce la intensificación a una parte mucho más pequeña de la población: sólo se ejecuta la búsqueda local sobre un 10 % de los cromosomas.

En principio, esta es una buena idea para intentar mantener la diversidad del algoritmo genético mientras que añadimos algo de intensificación.

Sin embargo, los resultados son incluso peores que con el anterior algoritmo. Excepto en la última base de datos, en la que la mejora es ínfima y poco significativa, este algoritmo se comporta peor que el generalizado.

Vemos así que, aunque la idea de aumentar la diversidad del anterior reduciendo la intensificación podía parecer positiva, en la práctica no ha dado los resultados esperados.

4.5. Algoritmo memético con búsqueda local elitista

	WDBC				Movement Libras				Arrythmia			
Particiones	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
Partición 1-1	99,2958	96,1404	60	33,3825	79,4444	78,3333	48,8889	44,0329	71,875	65,4639	53,9568	119,0935
Partición 1-2	97,8947	94,0141	60	32,4456	73,3333	71,6667	60	39,3526	74,7423	64,0625	52,8777	106,9755
Partición 2-1	97,1831	94,7368	53,3333	35,9236	76,6667	75,5556	58,8889	39,2133	73,4375	64,9485	57,9137	112,7924
Partición 2-2	98,5965	95,4225	50	34,6962	80,5556	71,1111	50	44,4735	69,0722	68,75	48,2014	126,8088
Partición 3-1	97,8873	95,0877	46,6667	35,485	75	78,3333	46,6667	45,229	69,7917	62,8866	53,2374	122,8596
Partición 3-2	97,8947	96,4789	43,3333	36,7941	72,2222	72,2222	52,2222	43,4023	71,6495	65,625	48,5612	120,9613
Partición 4-1	97,8873	96,8421	43,3333	36,105	77,7778	76,1111	45,5556	48,057	71,875	60,3093	53,5971	129,5047
Partición 4-2	97,8947	95,4225	23,3333	42,7348	73,8889	72,2222	54,4444	42,3835	71,6495	65,625	54,6763	108,8463
Partición 5-1	97,1831	96,4912	50	35,0131	74,4444	71,6667	56,6667	42,641	69,2708	63,9175	52,518	129,2749
Partición 5-2	98,9474	95,7746	53,3333	33,2209	72,7778	80	45,5556	46,5502	71,134	63,5417	49,6403	122,204
Medias	98,0665	95,6411	48,3333	35,5801	75,6111	74,7222	51,8889	43,5335	71,4497	64,513	52,518	119,9321

Cuadro 5: Datos del algoritmo memético con búsqueda local elistista.

Por último, veamos el algoritmo memético elitista, análogo al anterior pero eliminando la aleatoriedad del proceso al ejecutar la búsqueda local sobre el 10 % mejor de la población.

Este algoritmo sigue por tanto la misma idea que el último que hemos visto, pero intenta refinarla dejando que la búsqueda local actúe sólo sobre los mejores. Esto parece que da mejores resultados, quizás porque se dirige la intensificación a zonas del espacio de búsqueda más prometedoras, y no se desaprovecha esta potencia aleatorizando las zonas que mejorar.

Los resultados apoyan esta idea: este es el mejor de los tres algoritmos meméticos, excepto en la última base de datos. La mejora de la clasificación es sutil pero consistente en las dos primeras bases de datos.

4.6. Datos generales

Particiones	WDBC				Movement Libras				Arrythmia			
	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
3-NN	96,5196	96,2031	0	$3{,}5 \cdot 10^{-3}$	69,3889	74	0	$4{,}8 \cdot 10^{-3}$	62,5891	63,2689	0	0,0193
SFS	97,434	94,2701	85,6667	0,1986	76,7222	72	90,2222	0,9621	77,5677	68,292	97,8777	2,068
AM-(10;1.0)	97,68	95,5356	41,3333	37,5638	75,5556	74,2222	52,6667	42,9914	72,5306	65,131	50,6835	120,9622
AM-(10;0.1)	97,8907	95,0435	50,6667	34,971	74,7778	73,5556	50,8889	44,4334	71,5029	65,5488	49,7122	126,9701
AM-(10;0.1mej)	98,0665	95,6411	48,3333	35,5801	75,6111	74,7222	51,8889	43,5335	71,4497	64,513	52,518	119,9321

Cuadro 6: Datos generales.

En esta tabla se comprueba de forma más clara cómo actúan los algoritmos meméticos con respecto al algoritmo de comparación. Estudiemos cada base de datos por separado.

En la primera base de datos vemos que los tres algoritmos superan al de comparación, ganando entre ellos tres el elitista. Esto se debe principalmente a lo explicado en la última sección: el último algoritmo es el que más equilibrio presenta entre la diversificación y la intensificación.

En la segunda base de datos el comportamiento es similar: aunque en la tasa de clasificación en la partición de entrenamiento el algoritmo de comparación es el mejor, en la tasa de clasificación en la partición de test, que es la realmente importante, los tres algoritmos meméticos superan a SFS. De nuevo, aquí gana el memético elitista, ya que el tamaño de esta base de datos es parecida a la anterior y el equilibrio diversificación/intensificación da los mismos beneficios.

La tercera base de datos, como en todas las prácticas anteriores, presenta unos comportamientos diferentes a las dos previas. El tamaño de esta base es mucho mayor y sus datos están mucho más dispersos, como se puede comprobar comparando la tasa de clasificación del k-NN en esta y el resto de bases de datos, siendo aquí muchísimo menor. Estas características hacen que el SFS mejore a todos los algoritmos meméticos presentados. El equilibrio diversificación/intensificación del algoritmo memético elitista tiene aquí mucho menos poder, y de hecho es el perdedor entre los tres: esta base de datos necesita más diversificación para llegar a todas las zonas del espacio de búsqueda y, por tanto, es el memético aleatorio —que aun añadiendo intensificación no pierde diversidad como el generalizado— el ganador.

Con respecto a las tasas de reducción, los comportamientos de los tres algoritmos son muy parecidos en las tres bases de datos, consiguiendo tasas mucho menores que el SFS y todas sobre el 50 %. En este aspecto no hay gran diferencia entre los algoritmos, así que no podemos sacar conclusiones sobre su idoneidad basándonos en esta caractesística.

Como resumen a todo el análisis, podríamos decir que el mejor de los tres

algoritmos meméticos es el elitista, aunque antes de tomar una decisión hay que estudiar los datos que tenemos que procesar: si el espacio de búsqueda es potencialmente amplio, como en la última base de datos, entonces será más sensato usar el algoritmo aleatorio; si no, el elitista parece el más adecuado. En cualquier caso, y en base únicamente al estudio hecho aquí, que no es completo ni lo pretende, podríamos descartar el primero de los algoritmos si tenemos el tercero disponible, por añadir demasiada intensificación y no cuidar el mantener la diversidad, equilibrio que intentan conseguir todos los algoritmos meméticos.

Es importante destacar por último que cualquier conslusión extraída de este estudio debería ser analizada de nuevo con más casos de prueba y usando distintas componentes para los algoritmos, ya que es un análisis reducido que pretende únicamente dar una primera idea sobre la idoneidad de cada algoritmo para cada problema.